Environmental Concept for Engineering Software on MIMD Computers*

L. A. Lopez and K. Valimohamed
Department of Civil Engineering
University of Illinois, Urbana, IL 61801

Introduction

Novel computer architectures have been developed in an attempt to satisfy the performance specifications required to solve large nonlinear dynamic systems encountered in the sciences and engineering. In order to exploit the new architectures, it is often necessary to change the way we think about and write programs.

Several software options have been proposed to facilitate the programming of these sophisticated and complex machines [1-6]. However, existing solutions are rather restrictive or prohibitively expensive because they are not portable.

Recently, there have been some developments in environments and programming tools [7-10] that assist programmers in developing portable application code. However, these efforts have been restricted primarily to the control aspects of the program; i.e., describing to the computer when and how to do things in parallel. A key element in developing successful engineering systems is the management of large data spaces. Very little work has been done in this crucial area on MIMD machines.

The issues related to developing an environment in which engineering systems can be implemented on MIMD machines will be discussed. The problem is presented in terms of implementing the finite element method under such an environment. However, neither the concepts nor the prototype implementation environment are limited to this application. The topics discussed include: the ability to schedule and synchronize tasks efficiently; granularity of tasks; load balancing; and the use of a high level language to specify parallel constructs, manage data, and achieve portability. The objective of developing a virtual machine concept which incorporates solutions to the above issues leads to a design that can be mapped onto loosely coupled, tightly coupled, and hybrid systems.

* This work is supported by NASA Grant NAG-1-646

OBJECTIVE:

Define an MIMD Environment in Which We Can Implement Engineering Applications

CONSTRAINTS:

Solve Tomorrow's Problems as Opposed to Today's Problems Faster

Problems Will Have Large Data Spaces

Machine Independent Application Code

Portable Software Environment

OBJECTIVES AND CONSTRAINTS

The purpose of the project is to develop the concepts and to prototype a software architecture to support engineering systems on MIMD computers. A primary focus is to provide engineering programmers with a tool to express their problems without the need to become deeply involved in the concepts associated with parallel computers. The algorithm should transcend the computer architecture.

The governing philosophy of this research is to find methods to solve the largest problem possible within a given time frame rather than trying to solve existing problems faster.

The finite element method is the target application to be run on the environment. However, the support environment envisioned at this time is not limited to that application.

The kinds of problems envisioned for this environment are very large by today's standards. They will probably have data spaces in the 100 gigabyte range.

The investment in the next generation of programs will be so large that we will not be able to afford to start over with each new hardware concept. Therefore, both the application code and the environment will need to be machine independent.

GUIDING PRINCIPLES:

1) Application Program Concept Transcends
   The Computer Architecture

2) Data Objects:

   Are Primarily - Matrices and Hyper-Matrices

   Are Organized - Primarily Hierarchically

3) Application Can Express Concurrency at Multiple
   Levels

   a) Vectorization (Typically Mfgr)
   b) Concurrency using Precompilers

      i) Local software
      ii) Mfgr Provides Compiler
          Directives

   c) High Level Language Layer (2 levels)

GUIDING PRINCIPLES

The principles guiding us are:

> The application program concept transcends the computer hardware on which it executes. This is very important. One can express the finite element method independently of how the computer does the problem. This is not a new concept; FORTRAN statements are used to express algorithms rather than machine registers. Similarly, one can express the idea of natural parallelism in an algorithm independently of the method of achieving it.
>
> We propose that engineering programmers should be provided with tools to easily express various levels of potential parallelism in their algorithms without paying the consequences of fighting the details of implementation. It will be left to the combined hardware and software environment to decide what to do with the expressions.

To facilitate the data and memory management functions, the use of data objects will be necessary. Typical data objects will be matrices and hyper-matrices which are organized in a hierarchical form.

There are two types of parallelism to deal with - vectorization and concurrency. Vectorization is often handled automatically by the hardware and the compilers provided by the manufacturer. It should be transparent to the engineering programmer; we will assume its existence and will not deal with it directly in our research. Some researchers may argue that manufacturers do not do a good job of this. They will improve!

Concurrency is more difficult to detect, and is not performed automatically. There are several methods of specifying it in application systems today. A common solution is to use compiler directives. However, it is restricted to a particular manufacturer. An alternative is to use locally developed software like PISCES or the FORCE. They can then be ported from computer to computer without changing the application programs. Each of these methods is tied to the FORTRAN concept. Basically, they are sandwiched into what already exists.

An alternative is to develop a new high level language layer above FORTRAN. This is more difficult but eliminates the constraints of FORTRAN. Special constructs that are needed should be easier to implement. This method was used in the past in most large FEM (POLO/FINITE, DMAP/NASTRAN, DVS/ASKA, NICE/SPAR, ICES/STRUDL ...) systems. It is even more appropriate for MIMD computers.

GUIDING PRINCIPLES (CONT.):

4) Application Programmers Should Express Potential for Concurrency - Not How to Do it Concurrently

They Should Not Have To:

a) Assign Tasks to Processors
b) Perform Inter-memory Data Movements
c) Manage Memory or Data Directly
d) Perform Explicit Synchronization

GUIDING PRINCIPLES (cont)

Ideally, the user should only have to specify where the concurrency exists in the application program rather than explicitly defining how the application should be executed on a given machine. In the latter approach, for very large and complex problems, the application code tends to get "lost" in the details of: scheduling and synchronization of tasks; load balancing; inter-processor communication; managing the coherence of data objects in primary and secondary storage; and manipulating large data objects within a limited amount of memory.

Therefore, the programmer should not have to: assign tasks to processors; perform inter-memory data movements; manage memory or data directly; or perform explicit synchronization.

However, for improved performance, some interaction will be required from the programmer, the details of which will be described later.

MAJOR PROBLEMS TO SOLVE AND AUTOMATE:

1) Load Balancing - Tasks Implied by the Programmer Cannot Translate Directly to Tasks on the MIMD Architecture. That Would Tie the Program to the Architecture.

2) Data Coherence - MIMD Architectures Protect Single Words at the Hardware Level. Our Basic Unit is a Matrix. We Must Provide the Coherence Checks.

3) Memory Management - Classical Structural Mechanics Does Not Mention Global Memories, or Distributed Memories, or Hybrid Memories. They Should Not Appear in the Expression of the Solution.

MAJOR PROBLEMS TO SOLVE

Our objective is to develop concepts that hide most of the parallelism related activity from the programmer. In order to do this we have focused on solving (on paper) three major problems. They are automatic load balancing, data coherence, and data and memory management.

Automatic load balancing means that the programmer specifies only what can be done in parallel. The environment maps whatever is specified to the architecture in the best way possible by using the concept of relative granularity. Since the ideal task size for a processor will vary from one machine to the next, the tasks specified in the application program are interpreted and are executed in a manner which best achieves load balancing.

Data objects in large FEM systems are typically matrices and hypermatrices. Thus it becomes necessary for the the programmer, or in our case, the software environment to handle data coherence on large data objects. (Hardware data coherence is based on the granularity of a word size.) We believe that we can do this effectively via a combined queueing/data management/memory management system. Some of the ideas are clarified on the following slides.

Memory management has always been a problem for FEM programmers. The data space always exceeds the available memory (some undiscovered fundamental law of physics?). Memory management is not part of the classical FEM problem; neither is the concept of distributed memories; they should not explicitly appear in the FEM program either. Memory management must be handled automatically by the environment. In that way the programmer can concentrate on the conceptual FEM algorithm.

METHODS OF SOLUTION:

Are Evolutionary:

Precompilers (PISCES/FORCE)

Libraries (SCHEDULE)

High Level Language Layer (THAT'S US)

METHODS OF SOLUTION

The proposed solutions to problems relating to parallelism have been evolutionary, not revolutionary. In the past, FEM developers have used computer science concepts to develop environments with specific objectives. Precompilers were used in ICES. They are now being used in PISCES etc.. Libraries of routines called explicitly by the programmer were part of ASKA. They are also part of SCHEDULE - a system developed by Dan Sorenson et al at Argonne. High level languages were used in NASTRAN, POLO/FINITE, and NICE/SPAR and are proposed as a solution for the MIMD environment concept.

PROPOSED SOLUTION:

High Level Language Layer above Fortran

a) Define Data Structures

b) Define Processes for Operating on Data Structures

1) Simple Control Structures

2) Express Parallelism at Gross Level

3) Express Relative Granularity

4) Reference Data Objects and Corresponding Operations

PROPOSED SOLUTION

The combined machine independent automatic scheduling/DBMS/memory management proposed herein cannot be easily done by simply inserting additional commands into FORTRAN. Therefore we are investigating a language layer above FORTRAN that works in conjunction with FORTRAN. In that way we can provide a large number of features relating to large grain concurrency while taking advantage of whatever the manufacturer provides for fine grain concurrency and vectorization at the FORTRAN level. The latter will probably be supported with tools already available or under development at CSM.

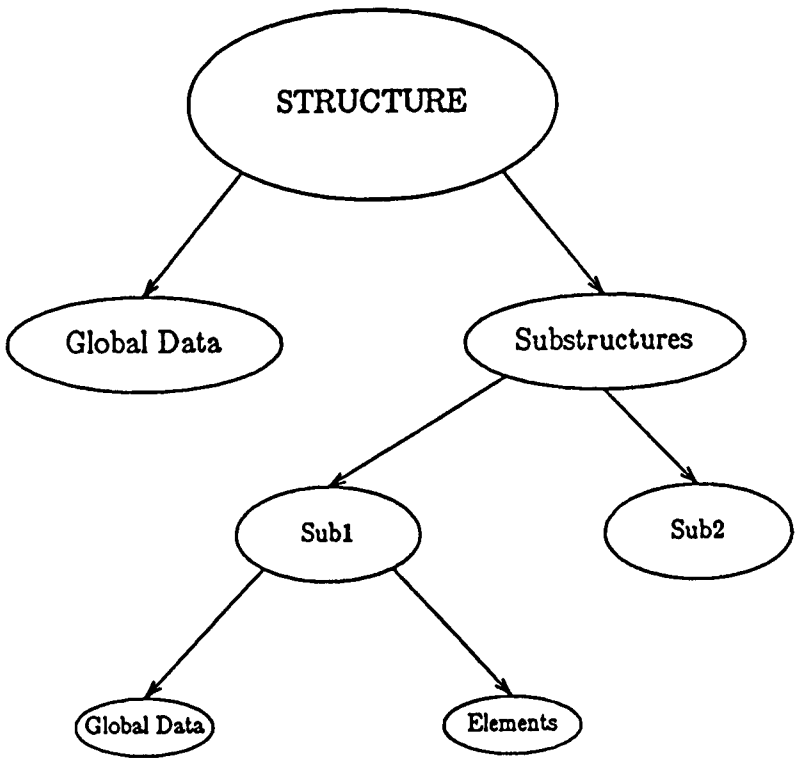
In the high level language, programmers will define data structures for data bases, and procedures that operate on the data. The system will utilize a compiled interpreter; i.e., the high level commands will be compiled to an IL (intermediate language). The IL code will be interpreted at run time. This technology has been used in the past to combine good performance with the flexibility of using an interpreter. The latter will prove very beneficial in helping programmers debug their systems, and in providing a mechanism for collecting statistics.

Interpreters can be slow. The idea is to provide a modicum of control structures and data referencing in the high level language and to combine it with FORTRAN code that has been compiled for the target computer. Using the analogy of [6], the high level language serves as a "skeleton" of the application program while the "guts" consists of FORTRAN code.

EXAMPLE OF HIGH LEVEL DATA DEFINITION:

```
STRUCTURES relation NUM_STRUCTURES
  attributes
    NUM_ELEMENTS
    NUM_NODES
    GEOMETRY     matrix          3 NUM_NODES
    TOPOLOGY     inverted_list   NUM_ELEMENTS
                                 NUM_ENTRIES
    STIFFNESS    hypermatrix ...

    SUBSTRUCTURES  relation  NUM_ELEMENTS
      attributes
        NUM_ELEMENTS
        NUM_NODES
        GEOMETRY   matrix    3 NUM_NODES
          .
          .
          .
```



EXAMPLE OF DATA DEFINITION

This slide demonstrates how a programmer might convert his concepts in data structure to something a computer could operate on. It is an example of how the DDL (data definition language) will appear to the programmer. Note that it is array structured and contains high level objects like relations, inverted lists, matrices, and hypermatrices.

EXAMPLE OF CONCURRENT TASK EXPRESSION:

PARALLEL_DO for s = 1,n_sub_structures **GRANULARITY Coarse**

SET_SUBSTRUCTURE (substructure_list(s, sub_structure:**r:g**))

PARALLEL_DO for i = 1,n_elem GRANULARITY Medium

CONNECTIVITY (substructures(sub_structure,incidences (i)):r:g, loc)

PARALLEL_DO for j = 1,n_sub_matrices GRANULARITY Fine

PUTMATRIX (stiffness (sub_structure, loc(j)):w:g:i, elstiff(i,j):r:l)

END_DO

.

.

.

CONCURRENT TASKS

This slide demonstrates how a programmer might express a small part of his algorithm. Note the multiple levels of parallel "DO" loops. The PARALLEL_DO combined with the relative GRANULARITY implies both desire for concurrency and the programmer's conception of the size of the tasks in the loop. The granularity statement is a concession - the programmer is getting into the details of parallelism. However, we think it is a necessary evil. It will permit the task scheduler to make intelligent decisions about how to execute this task structure on a given architecture. In order accomplish this, the system environment will use data that describes the configuration of the machine. This information is provided when the environment is initialized on a given computer.

In addition, this slide shows a number of data references. Some are followed with a sequence of one or more colons(:) and some additional data. The latter is optional (but useful) information that the programmer provides to help the system make intelligent decisions. For example a *:r* implies the data is being used in a read only mode. Other tasks may use it too. If it is *:w* it implies a need for write access. If neither is given, it defaults to write; the latter is safe but may lead to poor performance. For instance, a given iteration of a parallel do loop can be a task. Write access to an object will hold the object until the end of the task unless the programmer provides an *:i* for immediate release. For instance, a reference such as SUB(data:w:i) implies write access for the duration of the SUB operation, rather than the duration of the task in which the SUB operation is specified. It is similar to the fetch and add operation at the word level.

Each of the above attributes expands what the programmer needs to know about parallel computation. This is an admission of our failure to totally hide the parallel computational aspects of the problem. However, each attribute provides a simple mechanism to significantly improve performance.

IMPLEMENTATION CONCEPT:

All Data Definition is Compiled to an Internal Format

All High Level Processing Statements and Corresponding Data Object References are Compiled to an IL Code

A Copy of the System Resides on Each Processor. Using Monitor Concepts Each Processor Can Assume Master Status.

IMPLEMENTATION

As noted earlier, the high level language layer is most appropriate to meet all of the needs. Furthermore, interpretive data base management is much too slow to be practical. Consequently, all high level code and DBMS instructions will be compiled into IL code.

Data structures will be defined by the programmer using a high level language. The definition will be compiled into an internal format. This will allow the data manager to manipulate and examine data objects efficiently at run time.

There are a number of ways to implement parallel systems. After talking to individuals here and at Argonne we have concluded that a self-scheduling approach is most appropriate. In this scheme, each processor obtains 'work' (virtual process) from a global queue. This results in a dynamic load balancing scheme. Each processor will have access to the system functions. Using a monitor type approach in a shared memory machine, any processor can assume the responsibility of placing virtual processes onto the queue and removing them from the queue. Hence, one processor will not become a bottleneck by handling all messages and the queues. In a distributed memory architecture, the object that contains the monitor information can be passed around the network. A copy of the system resides on each processor.

In our original concept we had assumed that the number of processors available to a job would vary during the job; the system environment would obtain or release processors from the operating system as the work load varied in the job. However, most operating environments available today cannot support this. Consequently, the maximum number of processors must be specified when the job is initialized.

RUN TIME SUPPORT SYSTEM:

Is Invoked Automatically at Two Levels

- When Explicit Parallel Constructs are Encountered
- When Data Objects are Referenced

Maps Relative Granularity to Absolute Using Config Tables for this Architecture

Does Process Control and Load Balancing Via its Own Virtual Process and Sleep Queues

Uses a Segmented Multi-level Virtual Memory Manager to Handle All Three Types of Memories

Can be Implemented in Extended FORTRAN Environments like PISCES, SCHEDULE, or FORCE.

RUN TIME SUPPORT SYSTEM

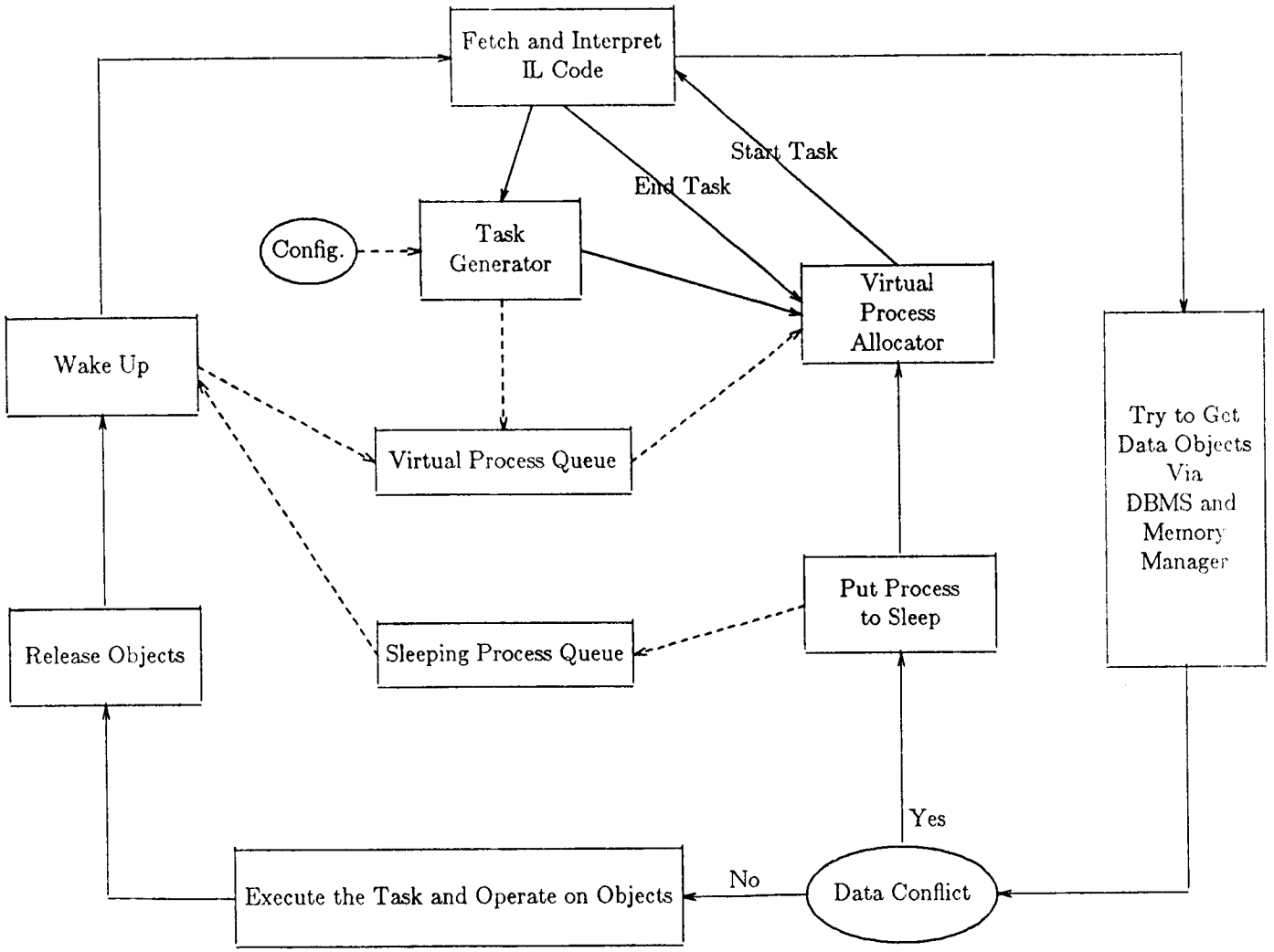
The run time support system is based on a virtual machine/virtual data base concept. It is invoked at two levels: when parallel constructs are encountered in the high level language; and, when data objects are referenced.

Virtual processes are created and then distributed among the real processors. A virtual process consists of one or more tasks (as defined in the high level language). A task consists of a set of HLL instructions which will perform a logical unit of work. Examples of a tasks are: an iteration of a loop body; or the execution of a subprogram (see next slide). A virtual process can be exist in several different states, e.g. 'sleeping', waiting or executing. The granularity of a task is specified by the programmer. It is defined using a relative scale. At run time, the sytem will try to create virtual processes that have an ideal granularity for the given machine. This is accomplished by mapping the relative task granularity into an absolute task granularity. Tasks are 'chunked' so that the virtual process will contain tasks of sufficient granularity to be executed on a processor. Configuration data (specified when the system was initialized) is required for the mapping procedure.

The run time system is based on a self scheduling scheme. Queues are used as buffer areas from which virtual processes can be manipulated. Dynamic load balancing will be achieved by having real processors 'take' work from these queues. Further information about the queues can be found on the next slide.

The virtual memory concept is implemented using a segmented-paging system. Data coherence is easier to maintain using segmentation, while manipulating data in the system favors objects which have a page-size granularity. Thus, each data object is identified as a segment, and can consist of one or more pages. Memory managers are used to maintain the coherence and distribution of objects in the various types of memory configurations (shown later).

The system will be developed such that it can interface with extended FORTRAN environments such as PISCES, SCHEDULE, or FORCE.



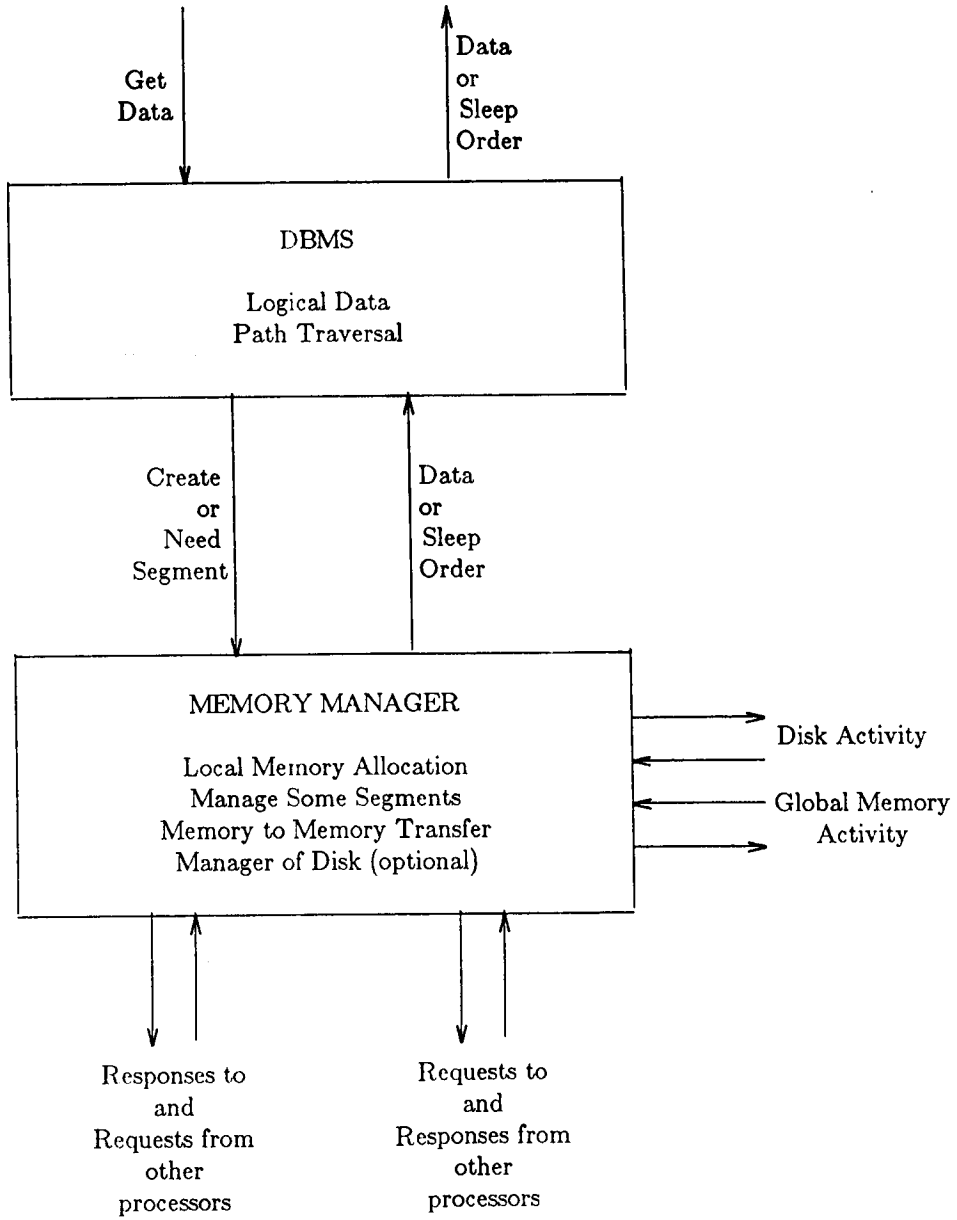
SYSTEM OPERATION

SYSTEM OPERATION

This slide shows the general parts of the run time system that resides on each machine. The interpreter at the top fetches instructions to be executed. If the request is for data objects, the block on the right is invoked. It may succeed or it may run into a coherence conflict. If successful, the HLL instruction in task requesting the data is executed. When an instruction or a task is completed, data objects are released, such as those associated with the end of a task or the *:i* (immediate release) mentioned earlier. If a data conflict occurs, the virtual process is suspended, it is placed in the sleep queue and remains there until the data object it was waiting for becomes available (i.e. the data object is released by another task), at which time the virtual processes is moved to the virtual process queue. The real processor which puts a virtual process to sleep gets the next virtual process from the virtual process queue. In this way, all the processors are kept busy.

When the IL code indicates that a new task has been specified, the machine configuration information and the directives provided by the programmer are used by the task generator to determine the appropriate granularity of a virtual process. The task generator will place the virtual process in a queue which is shared by the virtual process allocator.

The Virtual Process Allocator (VPA) is responsible for assigning work (virtual processes) to the real processors. Virtual Processes are released from the queue only when a request is made for some work and the synchronization instructions permit the *VPA* to do so. If the queue is empty, control is passed to the task generator to fill the queue. After a real processor has received its work, control is passed to the interpreter of that processor. After the instructions for that virtual process have been executed, the processor will ask for more work. This process will continue until the job is completed.



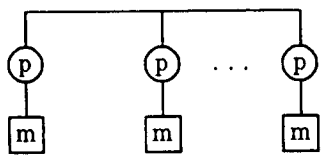
DBMS/MEMORY MANAGER INTERACTION

OBJECT AND MEMORY MANAGEMENT

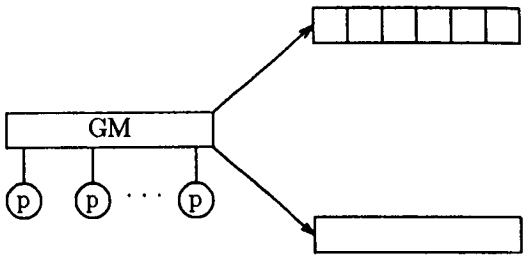
The data and memory management aspect is the most significant part of the system. It is what makes this concept totally different from other approaches; it makes parallel processing a virtual problem.

This slide shows that there are two major parts involved. The DBMS/IL code instructions are used to trace paths through logical data tables and objects. Each object in the hierarchy is requested from the memory manager. Internally the system must have a message passing architecture. On some hardware configurations these messages will be transmitted over the network; on others they may just degenerate into changing the values of variables that are shared by other parts of the software system.

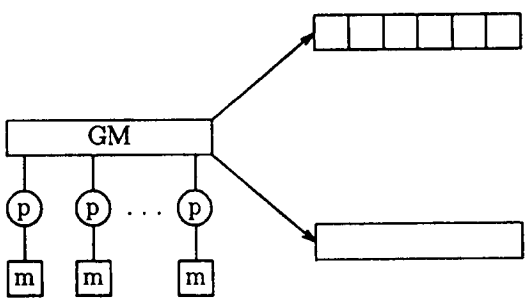
The memory manager is responsible for ensuring that the objects requested by the processor are made available in the processor's memory (local or global). Various memory configurations are shown on the next slide.



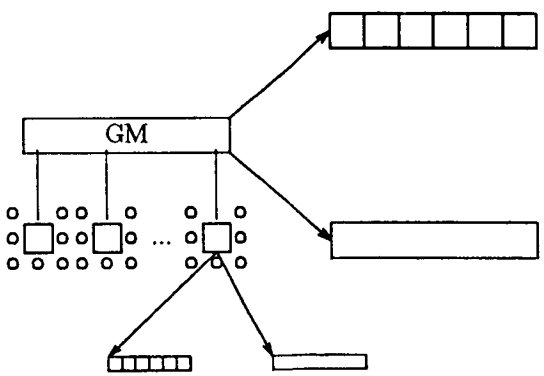
(a) DISTRIBUTED



(b) SHARED



(c) HYBRID CASE 1



(d) HYBRID CASE 2

KEY

- m — memory module
- p — processor
- set of cluster processors with cluster memory
- partitioned shared memory
- unpartitioned shared memory

GM = Global Memory

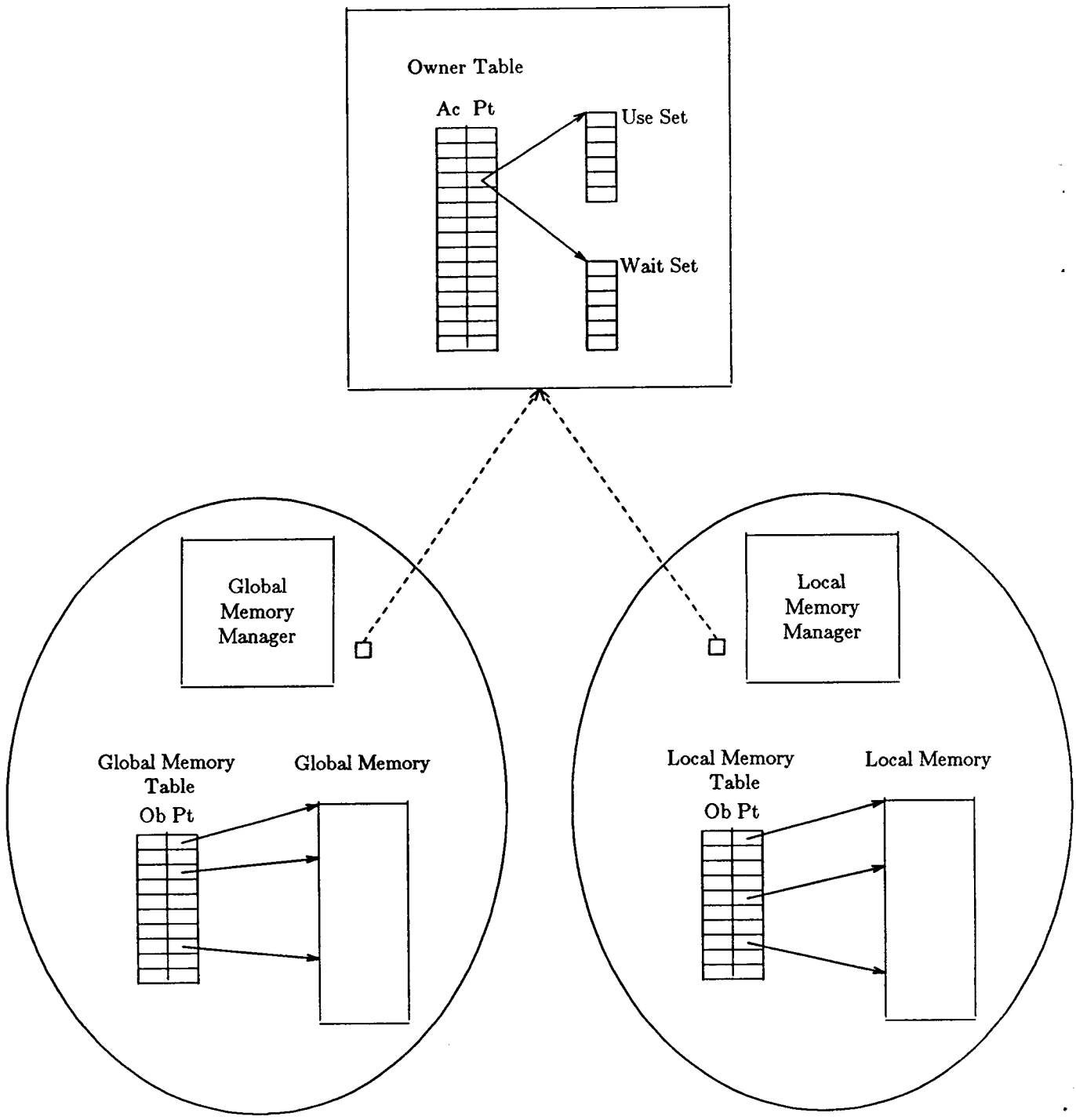
MEMORY CONFIGURATIONS

TYPICAL MEMORY CONFIGURATIONS

This slide shows the types of memory configurations we have considered thus far: Global; Distributed; and, Hybrid. All the hardware architectures that we have examined fall into one of these categories. Global memory (GM) configurations are always shown with two alternatives - either a single block or subdivided into individual blocks that are logically associated with a processor. It is unlikely, but possible, that this type of subdivision (via software) may work better than the single global memory approach. It appears that it will be a function of the particular application and how memory conflicts arise.

Hybrid case 1 is a representation of the ETA or the other reconfigurable machines such as the FLEX. Hybrid case 2 is a representation of the CEDAR machine being developed at the CSRD at Illinois. It consists of clusters of Alliant processors interconnected to a global memory. Since Alliants also use a global memory (referred to as *cluster memory* in Cedar terminology), global memory is shown at two levels in the figure. Each global memory is then shown as having two possible configurations - segmented and unsegmented.

Since combinations of memory types exist on some architectures, it necessary to have all types of managers or managers with attributes of both shared and distributed systems. If necessary, switching between two types of memory managers is done dynamically, and is based on the kinds of pointers (location) of the desired data.



Memory Management Tables

MEMORY MANAGERS

Memory managers insure that the desired object is present in memory for a processor when it needs it. There are two basic types of memory managers. One manages the memory on the processor itself. This system, called a local manager, sees its own memory directly. It controls what is in it, and where to put things in it. It needs no permission to look at data etc... The one exception is when its memory is used (partially) to store global tables. That will occur in totally distributed systems; in which case any information that is global to all processors is stored on one processor's memory.

The other type of memory manager that resides on all processors is a global manager. It is responsible for operating on either of the two types of global memory shown earlier.

Note that in the CEDAR machine there are two levels of global memory managers. Those at the cluster level have some of the attributes of the distributed memory managers - since clusters are distributed. Configuration tables will be used to determine which types of memory managers are to be used for a a given configuration.

Memory management for data objects is accomplished via ownership tables shown at the left. Each data object currently in use has an entry in that table. We estimate that, for the 100 gbyte problems, this table will contain about 5 megawords. It will be distributed evenly on a distributed memory system and stored in one memory on a global memory system. Since every object will not be in use at any given time, the full size of the table will not be required.

In the owner table we have access locks, information about which processes are currently using the object, and which processes are waiting for the object. (See slide on system queueing.)

There are some interesting problems with maintaining the use and wait sets. These result from our attempt to hide parallelism from the programmer. It is not always clear what he had in mind. In some instances combinations of entries are clear errors. In other instances they may be a result of the "system working ahead" or latency in the network; i.e., allowing multiple tasks to begin when it is known that there will be conflicts that can be resolved dynamically. If we don't allow this kind of activity, there will be unnecessary constraints on the application that may lead to serious degradation in performance. The *:i* flag in the high level language was introduced as a way for the programmer to inform the system memory manager that the apparent conflict was anticipated. All others will generate warnings.

FUTURE:

Continue to Refine the Concept

Begin Implementing a Prototype

FUTURE

We will continue to refine the model. As we find problems, we will attempt to determine if they are a result of hiding too much of the parallel problem from the programmer. If so we extend the model of what a programmer sees in the high level language. If it is simply a mechanical problem related to the system design, or to some new type of architecture (we think we can handle chordal rings) we make the appropriate additions to our low level system concept.

This Spring we expect to begin a prototype of the memory management system. It will be something that we can use in simulations while the higher level components are developed. It may also be of use to others in the CSM family.

REFERENCES

[1] Almasi, G.S., "Overview of parallel processing", *Parallel Computing*, Vol 2 (1985) pp 191-203

[2] Hockney, R.W., "MIMD Computing in the USA - 1984", *Parallel Computing*, Vol 2 (1985) pp 119-136

[3] Gajski, D.D. and J-K Peir, "Essential Issues in Multiprocessor Systems", *IEEE Computer* Vol 18, No 6., June 1985. pp 9-27

[4] Kuck, D.J. et al "Parallel Supercomputing Today and the Cedar Approach" *Science*, Vol 231, Feb 28 1986, pp 967-974.

[5] Snyder, L. "Type Archictectures, Shared Memory and the Corollary of Modest Potential" *Department of Computer Science, FR-35* University of Washington. Seattle, WA 98195 TR 86-03-04

[6] McGraw, J. and T.S. Axelrod, "Exploiting Multiprocessors: Issues and Options" *UCRL-91734 preprint* Lawrence Livermore National Laboratory Oct. 31 1984

[7] Jordan, H.F., M.S. Benten, and N.S. Arenstorf, "Force User's Manual" *Dept. of Computer and Electrical Engineering, University of Colorado* October 1986

[8] Pratt, T.W., "PISCES: An Environment for Parallel Scientific Computation" NASA-ICASE Rep. No. 85-12

[9] Dongarra, J. and D. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," *Tech. Memo. 86, Argonne National Lab.*,, November 1986

[10] Padua, D.A., V.A. Guarna Jr. and D.H. Lawrie., "Supercomputing Environments" *CSRD Report No. 679* Center for Supercomputing Research and Development, University of Illinois, Urbana, IL. June 1987